The ExoVM System for Automatic VM and Application Reduction

Ben L. Titzer
UCLA Compilers Group
titzer@cs.ucla.edu

Joshua Auerbach
IBM T.J. Watson Research Center
josh@us.ibm.com

David F. Bacon
IBM T.J. Watson Research Center
dfb@watson.ibm.com

Jens Palsberg
UCLA Compilers Group
palsberg@cs.ucla.edu

Abstract
Embedded systems pose unique challenges to Java application developers and virtual machine designers. Chief among these challenges is the memory footprint of both the virtual machine and the applications that run within it. With the rapidly increasing set of features provided by the Java language, virtual machine designers are often forced to build custom implementations that make various tradeoffs between the footprint of the virtual machine and the subset of the Java language and class libraries that are supported. In this paper, we present the ExoVM, a system in which an application is initialized in a fully featured virtual machine, and then the code, data, and virtual machine features necessary to execute it are packaged into a binary image. Key to this process is *feature analysis*, a technique for computing the reachable code and data of a Java program and its implementation inside the VM simultaneously. The ExoVM reduces the need to develop customized embedded virtual machines by reusing a single VM infrastructure and automatically eliding the implementation of unused Java features on a per-program basis. We present a constraint-based instantiation of the analysis technique, an implementation in IBM's J9 Java VM, experiments evaluating our technique for the EEMBC benchmark suite, and some discussion of the individual costs of some of Java's features. Our evaluation shows that our system can reduce the non-heap memory allocation of the virtual machine by as much as 75%. We discuss VM and language design decisions that our work shows are important in targeting embedded systems, supporting the long-term goal of a common VM infrastructure spanning from motes to large servers.

Categories and Subject Descriptors C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—*run-time environments*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*.

General Terms Performance, Design, Languages, Verification.

Keywords pre-initialization, embedded systems, persistence, dead code elimination, static compilation, static analysis, VM design, VM modularity, feature analysis

1. INTRODUCTION

Developers have long recognized the advantages of virtual machines for embedded systems; in fact, the development of Java was originally motivated by the need to develop portable software for cable set-top boxes. Embedded platforms such as sensor nodes and cell phones are orders of magnitude smaller than desktop systems, making resource limitations of paramount importance to developers of both applications and virtual machines. The limitations of such devices have slowed the adoption of Java and other modern languages that require a large runtime system. We believe that this is because a quality Java virtual machine that supports dynamic class loading, JIT compilation, advanced garbage collection, and the complete Java language specification and accompanying class library is a dauntingly large piece of software. A number of specialized embedded virtual machines have been developed [5][7][13][15] that target embedded systems and have investigated various subsets of the Java specification, and a number of standards have arisen, for example, the Connected Limited Device Configuration [4].

To combat the space limitations of these embedded domains, researchers have investigated a number of techniques, including heap compression [3], class file reduction [16], and VM specialization [7]. Many of these systems begin with a custom Java virtual machine implementation; i.e. a virtual machine specifically designed for small footprint as opposed to feature completeness or performance. For example, VM* [6] is an extremely bare-bones customizable Java interpreter with a very minimal class library targeting mote class devices. KVM [13] is a specialized virtual machine with custom class libraries targeting embedded devices with at least 192kb of RAM.

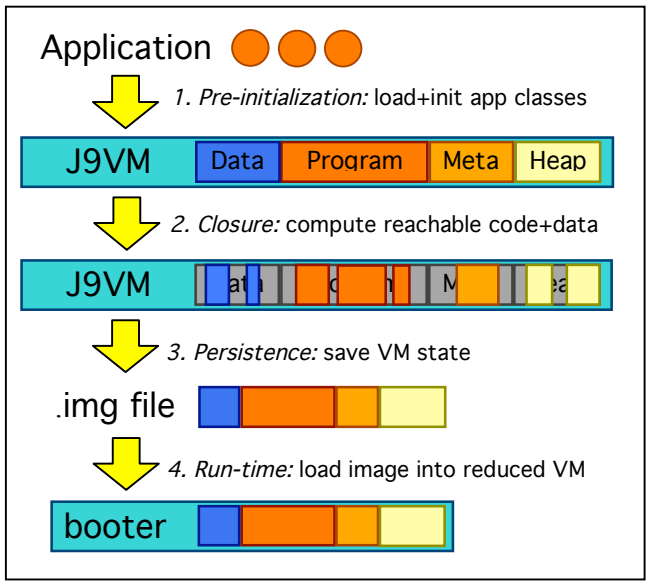
While developing a customized virtual machine and class library for an embedded system domain has its advantages, it also has important disadvantages. First, though a small VM is comparatively less engineering effort than a fully featured one, software development and maintenance effort is inevitably duplicated. Secondly, both incremental improvements and significant advancements in the state of the art in implementation technology cannot be automatically utilized in the custom VM. Thirdly, evaluations of research ideas and implementation techniques inevitably have narrower scope because results are not immediately comparable across domains that do not share a common virtual machine infrastructure.

The ExoVM approaches this problem with the following philosophy:

> *Reuse* existing VM technology; make the program as *static* and predictable as possible; and *include* only what is necessary on a per-program basis.

The starting point of the ExoVM system is to reuse a complete JVM implementation and Java class library. This could be any industrial or research system that supports a sufficient feature set. In our work, we chose an industrial strength virtual machine, IBM's J9 VM, but we believe that the general techniques described here could be applicable to any virtual machine.

PLDI'07 June 11–13, 2007, San Diego, CA, USA.



Overview of ExoVM architecture

The second part of the philosophy is to limit the dynamism of the program, or, almost equivalently, to restrict our attention to programs that are largely static. While Java has a number of dynamic features, in the embedded domain, many developers and applications already assume a closed world scenario. Applications are generally statically configured and then deployed onto the device; execution on the device often does not require dynamic class loading, reflection, etc. We believe this philosophy to be sound for a large, important set of embedded programs.

The key insight in this paper is to recognize that the second and third philosophical points allow *pre-initialization*, *closure*, and *persistence* over both the program and the virtual machine implementation together. Normally, the virtual machine builds data structures for itself and the program during VM startup, application loading, and also lazily during application execution; with pre-initialization, all of these data structures are built *before* running the program. Closure is the process of computing the reachable portion of the complete system (both the VM and application) over any execution, including the program and VM's code and the pre-initialized data structures. Persistence is the process of copying these data structures from the pre-initialized environment to the environment in which the program will run.

The ExoVM approach to each of these is to i.) perform pre-initialization of the program and VM by loading the program into the fully featured virtual machine and running the static initializers of the Java classes; ii.) compute the closure using *feature analysis* to analyze the program and VM simultaneously; and iii.) persist the data structures computed by the closure process into an image file that can be loaded by a specialized boot VM that elides subsystems that are not needed to run the program.

Our work is similar to previous work by Courbot and Grimaud [5], who built a customizable VM for the purpose of pre-initializing and reducing embedded programs prior to deployment. While the approaches share the same general philosophy, the work we present in this paper has three key differences. First, we begin with an existing industrial virtual machine implementation and class library, because we believe in a larger goal of reusing the same VM infrastructure for all classes of devices. Secondly, we do not modify the virtual machine or its internal representation of program quantities in order to support initialization or reduction, but instead build our analysis on top of the virtual machine without disturbing its implementation. Thirdly, we have developed a constraint-based program analysis that allows our system to approximate the implementation of native code for the purpose of analysis and therefore express the interdependencies between the virtual machine, the class library, and the class libraries' native code in a seamless framework.

The starting point for the system presented here is a development configuration of the J9 virtual machine that does not precisely correspond to a particular IBM product. We based our ExoVM implementation on the CLDC 1.1 MT version of J9 and additionally included some minimal Java reflection support that is required to implement ExoVM pre-initialization and closure computation. We developed the system on Linux x86, but the ideas and results reported here should transfer to similar configurations on different architectures, e.g. 32-bit ARM. We studied two variants of this VM: one using the CLDC class library (j9cldc, approximately 190kb), and another using a much larger class library that approximates the J2SE 1.4 (j9max, approximately 1.6mb).

2. FIXED AND PROPORTIONAL COSTS

The memory footprint of a Java application is not only comprised of its own code and data and that of libraries, but also that of the virtual machine. We can classify the cost of the virtual machine into two main quantities: a *fixed cost* and a *proportional cost*. The fixed cost corresponds to the VM's code and static data structures that are independent of the application, such as a garbage collector, runtime class loading mechanism, interpreter, JIT compiler, etc. The proportional cost corresponds to program's code and heap—e.g. the internal representation of its classes, bytecodes, dispatch tables, compiled code, object type information, method exception tables, Java objects, etc.

For many embedded applications, the fixed cost of the JVM runtime system and its data structures may dwarf the size of the application. For example, the j9cldc VM executable has more than 600kb of native code, 40kb of static data, and 190kb of Java classes, while none of the 6 EEMBC benchmarks (Section 6) requires more than 120kb for its class representations, and 5 of 6 execute successfully with a heap less of just 128kb. We believe that this hampers the development of small Java applications for small devices.

For larger applications, the fixed cost of the VM becomes less of an issue, and eventually the proportional cost will dominate. Thus an ideal situation would be a small fixed cost for small, simple programs and a proportional cost that is related to the size and characteristics of the application so that simplifications and reductions of large programs produce predictable reductions in total footprint.

Our observation is that the virtual machine's fixed cost is not (or should not be) as fixed as previously thought, and the VM can be divided into fine-grained pieces of functionality that can be related to features in the Java programming language. Dividing the VM along feature lines allows costs that were previously fixed to become proportional to the feature usage of the program. Automated program analysis can then produce the set of features used in a particular application and therefore allow a customized Java VM with a smaller fixed cost to run the application.

3. PRE-INITIALIZATION

Many large programs have sophisticated initialization routines that build complex data structures for use throughout the life of the program. In the case of a Java virtual machine, there are data structures to represent and manage the program and the program's state, including threads, Java classes and methods, locks, the garbage collector, JIT compiler, the Java heap, etc. The insight of pre-initialization is that these complex, often long-lived data structures that are normally built at the beginning of the program execution can instead be built offline and saved for use when the program begins execution. This saves the cost of the initialization routines in both startup time and code footprint.

Our first goal is to reuse an existing virtual machine rather than build a customized virtual machine. To support pre-initialization, the data structures needed by the VM must be built offline somehow and saved. We began studying our fully featured VM and soon discovered that the mechanisms that build and maintain internal data structures both at startup and throughout the execution of the program (e.g. resolving and loading a class) are remarkably complex. Our first approach was to attempt to replicate the construction of these VM data structures in an offline manner, but this foundered due to the complexity of trying to replicate the effect of the startup routines. We quickly discovered that a more elegant solution is to simply reuse the existing initialization routines by running them to a consistent state, and then taking a snapshot.

The ExoVM system implements this solution by loading the program into the fully featured virtual machine using the standard startup and loading routines in a non-intrusive manner. This causes the virtual machine to initialize itself to a state that is ready to begin executing the program. In particular, the VM has already built the internal representation of the first of the program's classes and methods as well as parts of the class library. The VM has already allocated threads, allocated some initial Java objects, and resolved important Java classes needed in the internal implementation of certain language features. Thus the ExoVM analysis system has a complete picture of the initial data structures that are required to begin executing the program. Pre-initialization continues by forcing the VM to load rest of the application classes (which would normally be dynamically loaded during application execution), which causes it to build the internal representations of these classes.

3.1 Class Initializers

In Java, a class may define an optional *class initializer* (also called a static initializer), a static method that is executed upon the first use of the class while the program is executing. While lazy initialization gives rises to some semantic problems (e.g. nondeterminism in initialization, exceptions in initializers, cyclic dependencies, and dynamic incompatible class change exceptions), in this paper, we are concerned with program analysis and footprint, and this mechanism can be particularly troublesome.

The dynamic resolution of class, method, and field references in Java code has definite implementation costs. First, it requires that the constant pool references include the metadata needed for dynamic resolution, including the string names of methods, fields and classes. Second, dynamic resolution may trigger class loading and initialization. Third, the VM must also maintain more metadata for every declared class, field, and method to anticipate any new references in the future. Fourth, resolution mechanisms inevitably include hash tables and other such fast search data structures. Our view is that while dynamically loading application classes may reduce the average case footprint for some applications, the basic classes in the Java library have dependencies that trigger large numbers of classes to be loaded and initialized (many of which are never used by the program) which leads to the effect of a large fixed JVM cost.

We consider dynamic resolution and initialization of classes as unwarranted complexity and resource consumption, which lead us to explore the implications of changing the model according to our original design philosophy of making the program more static. Thus, the ExoVM aggressively executes all class initializers for the live classes of the program and resolves all constant pool references to classes and methods as part of the pre-initialization phase.

Changing the model has advantages as well as disadvantages. First, it ensures that class initializers will not need to be executed at runtime, which allows their code to be removed. Second, no dynamic resolution of class, method, or field references will occur, so the metadata that is needed for dynamic resolution can be removed, and the mechanism can be removed from the VM. Third, this allows a program written with the model in mind to pre-allocate needed data structures in its static initialization routines, which are discarded before runtime, yielding a staged computation model closer to that proposed in [12].

One disadvantage of this approach is that it subtly alters the semantics of Java's class initializers, which some programs may depend on. Also, eager initialization could trigger the execution of routines that may not be triggered at runtime, which might allocate large data structures that waste space, destroy the state of other classes, and generally interact in unintended and unpredictable ways. However, we believe that most programs for this domain do not depend on the order or laziness of initialization. For example, in the EEMBC benchmark suite, only one program, `Parallel`, appears to do significant computation in its class initializers. This initializer does not depend on other classes, but simply allocates and initializes a static matrix of data that is used during the benchmark. Moreover, we believe that the closure technique described in the next section will automatically remove many data structures that are allocated by the initialization phase but are unused at runtime.

4. CLOSURE AND FEATURE ANALYSIS

To ensure the smallest possible program footprint, we would like to automatically compute the smallest set of classes and methods that are reachable over any execution of the program. There are a number of whole-program techniques to address this problem, including RTA [2], CHA [6], RMA [12], and flow analyses such as 0-CFA, as well as whole-module analyses such as that used in Jax [10]. All of these techniques share a common conceptual approach to the problem, beginning at some entrypoint method(s) in the program and building a static call graph that approximates the reachable code in the program. If a closed world assumption is made, code that is not reachable can be safely removed. If an open world is assumed, constraints can be added to prevent unsafe removal of possibly live code.

In the ExoVM system, we must compute reachability over not only classes and methods in the Java program, but over the initial Java heap as well as the data structures and code in the virtual machine. Therefore our analysis builds on both RTA and RMA and extends the class of whole-program, closed world techniques

that include live heap objects in the analysis. While RMA operates on the live heap of a program and its code together and removes code, objects and fields of objects, we need three new types of constraints that relate entities at the program level to entities at the implementation and VM level.

4.1 Feature Analysis

Now we will discuss *feature analysis*, which extends the traditional approach of analysis over program entities to include analysis of entities that are the explicit implementation of language features within the virtual machine. We will use the term *entity* to refer to a single data structure instance, Java object instance, Java method, string constant, or VM native method that consumes either code or data space. Unfortunately, in discussions of programming languages, the term *feature* is perhaps the most loosely used and most ill defined. However, we will use the term *feature* to refer to the members of or operations on entities.

These definitions have the effect that we restrict our attention to entities and features that have an isolatable implementation in the virtual machine. Our analysis makes entities in the virtual machine explicitly analyzable and will only include entities in the final program image if they are reachable through feature usage in the program.

Focusing in this way on entities and features with identifiable implementation artifacts, we can reason more concretely about the language in terms of these implementation artifacts. For example, a large, coarse-grained service might be garbage collection—it has a well-defined set of entities in its implementation that require metadata about classes, objects, methods, and threads in order support precise collection. Another example might be the ability to invoke the `getClass()` method on an object, which allows inspection of the run-time type of an object. This feature also has an identifiable implementation; the VM has data structures that represent classes that are exposed to the programs that call this method. Another example is the use of the `Class.forName()` static method; this method's implementation requires the VM a mapping between string names and class representations, as well as the ability to search for a class if it is not already loaded. If the program does not invoke this method at any point, then the data structures corresponding to implementing this feature can be removed. Other, finer-grained examples are floating point arithmetic, explicit casts, synchronization operations, weak references, JNI, reference arrays, static initializers, and exceptions.

Many of these features correspond almost directly to Java bytecodes, and some correspond to Java library methods and classes. But other features become apparent after some study of a virtual machine implementation, such as the ability to search for a method by its name in a particular class, or to resolve constant pool entries, which though they do not have a direct language expression, are demanded by the implementation of other features. For example, the ability to search for a class by its name is necessary for the VM to resolve some internal Java classes such as the exception classes.

The key idea behind feature analysis is that by treating these VM data structures as first class entities in the closure process (just like Java classes, objects and methods), the analysis can express the implementation of the language as members and features of these entities. Reachability over VM data structure instances then becomes analogous to the familiar notion of reachability over heap objects; an entity is only reachable if it is referred to by another reachable entity through a feature. If an entity is not reachable through a chain of feature uses in the program and the virtual machine, then it can be safely removed from the image.

4.2 Constraint-based Analysis

Constraint-based program analyses separate the specification of a correct solution to a program analysis problem from the implementation of the algorithm that computes the best solution. For example, in a program analysis problem such as flow analysis or pointer analysis, the primary goal is to compute sets of program quantities, such as *"what variables may this pointer refer to over any execution of the program?"* or *"what method implementations are reachable at this call site in the program?"*. Constraint-based analyses usually have the property that there is always a default, correct, but overly conservative solution such as *"this pointer might point to anything"*. The art of getting a good and verifiably correct solution to the analysis problem is deriving a rule set that describes the minimal properties of a correct solution. Once the constraint system is set up for a particular program, a general constraint solver can compute the least solution to the constraints, giving the most precise answer.

4.3 Entities and VM Types

The overall goal of our analysis is to compute the set of live entities needed to implement the program, both at the Java level and at the VM level. Each entity in our analysis has an associated type, and each entity type has an associated set of live instances, with the overall analysis result being the union of all entity sets. An entity is considered live and should be included in the closure if it is contained in its associated entity set. Our analysis models Java-level entities such as methods, classes and objects in a manner that is similar to RMA [12]. To simplify the constraints, Java methods with implementations have type `method`, classes have type `Class`, and each object instance's type is its dynamic Java type. Note that each of these Java-level entities may have one or more associated VM-level entities, not all of which may be ultimately considered live.

In addition to the Java entities of the program, our implementation models 24 different types of VM data structures that are listed in Figure 1. Among these types are: `VMNative`, which models the native code implementations of java methods such as `Object.hashCode()`; `VMClass`, the in-memory representation of a Java class; `VMMethod`, the in-memory representation of a method; `VMROMClass`, the on-disk and in-memory representation of the read-only portion of a Java class such as string names, the

VMNative	VMStackWalkState
VMJavaVM	VMHashTable
VMClass	VMMemorySegment
VMArrayClass	VMMemorySegmentList
VMClassLoader	VMPortLibrary
VMROMClass	VMThreadMonitor
VMMethod	VMJavaLangString
VMROMMethod	VMJavaLangThread
VMConstantPool	VMInternalVMFunctions
VMROMConstantPool	VMMemoryManagerFunctions
VMITable	VMInternalVMLabels

Figure 1. A list of the VM types that we model in our analysis. Each type has a list of associated features that are used in implementing various language features. The `VMNative` entity models implementations of Java native methods from the class library that are supplied by the VM.

constant pool, declared methods; VMThread, a representation of a Java thread; and the all-important VMJavaVM data structure, which contains pointers to important classes, the heap, collections of classes, threads, and at least a dozen other subsystems.

Each pointer field within a native data structure is modeled as a feature. This allows fine-grained precision in the analysis of the data structures of the VM. Our analysis models dozens of features for these types; space limitations preclude a complete list.

4.4 Constraint Sets

Our analysis uses two kinds of sets. The first kind of set, an **E** (entity) set, contains live entities such as Java objects, VM data structure instances, or Java method implementations. For example, for a Java class C, the set $\mathbf{E_C}$ represents the set of all reachable objects of exact dynamic type C in the initial heap.

The second kind of set is an **F** (feature) set, which is a set that contains the used features of a particular type. The set $\mathbf{F_C}$ for a Java class C contains the declared fields and methods of C that have been used explicitly within the program. Similarly, the $\mathbf{F_T}$ set for a VM type T contains the declared fields of T that are used by the program and the VM. Consider the VMMethod type. It has declared fields name and signature that reference UTF8 strings. These fields are modeled as features of the VMMethod type, and if the fields (features) are used, then they will be added to the $\mathbf{F_{VMMethod}}$ set. Further constraints will ensure that the strings to which these fields refer will be included in the closure.

There is one $\mathbf{E_C}$ set and one $\mathbf{F_C}$ set for every Java class C in the program and one $\mathbf{E_T}$ set and one $\mathbf{F_T}$ set for every type T of VM data structure types. To simplify the number of different types of constraints, our analysis models a Java method implementation (i.e. a method that contains code) as an entity of type method, and the set of all reachable method implementations with $\mathbf{E_{method}}$.

4.5 Constraint Forms in Feature Analysis

Our analysis generates 8 types of constraints. Some of these constraint forms should be familiar to readers who have prior experience with RTA [2] or RMA [12].

(1) Base case for entities: expresses initially reachable entities. If an entity e of type T is present at the beginning of the program execution, for example the main method, then e is reachable.

$$e \in E_T$$

(2) Call site: analyzes call sites in the code of reachable methods in the program. For each method M and each call site e.p() in the code of M, where the static type of e is C, we have the constraint:

$$M \in E_{method} \Rightarrow p \in F_C$$

(3) New object: analyzes allocation sites in the code of reachable methods in the program. We use $dummy_C$ to denote a dummy entity of type C. For each method M and each new C() in the code of M, we have the constraint:

$$M \in E_{method} \Rightarrow dummy_C \in E_C$$

(4) Feature use: approximates the result of using a feature of a type by using the feature on *all live instances* of that type. Specifically, if the entity e_0 of type S is live, and the feature f of type S is live, then the entity referred to by e.f is also live:

$$f \in F_S \wedge e_0 \in E_S \Rightarrow e_0.f \in E_{typeof(e^0.f)}$$

(5) Subtyping: establishes the relationship between used features in a supertype to the used features in a subtype. Specifically, for types S and T in the Java program, where S is a subtype of T, we have the constraint:

$$F_T \subseteq F_S$$

(6) Feature implication: expresses cases where the use of a feature entails that some other feature is also used. Specifically, for a type S with feature f, and a type T with feature g we may have a constraint of the form:

$$f \in F_S \Rightarrow g \in F_T$$

(7) Entity implication: expresses cases where the reachability of one entity implies the reachability of some other entity. Specifically, for an entity d of type S, and another entity e of type T, we can have constraints of the form:

$$d \in E_S \Rightarrow e \in E_T$$

(8) Entity implies feature: expresses cases where the reachability of one entity entails the use of a feature of some other type. Specifically, for an entity e of type S, and for a type T with feature f, we may have the constraint:

$$e \in E_S \Rightarrow f \in F_T$$

The constraints (1), (2), and (3) are basically equivalent to rapid type analysis, which maintains a set of possibly instantiated classes RTA_C and a set of reachable method implementations RTA_M. We can take this view if we consider the existence of $dummy_C$ in E_C is equivalent to C being in the live set RTA_C maintained in RTA. However, constraints (4) and (5) extend this basic view with live entity sets that are similar to those maintained in the RMA [12] analysis. The key insight is that the new constraints (6), (7), and (8) extend the power of the analysis even further, allowing us to specify per-language and per-VM constraints that relate Java entities to their implementation and vice versa.

(a) $fillInStackTrace \in E_{VMNative}$
$\Rightarrow dummy_{[I} \in E_{[I}$

(b) $fillInStackTrace \in E_{VMNative}$
$\Rightarrow classSegmentList \in F_{VMJavaVM}$

(c) $startThread \in E_{VMNative}$
$\Rightarrow run \in F_{java.lang.Runnable}$

(d) $startThread \in E_{VMNative}$
$\Rightarrow J9VMInternals.threadCleanup \in E_{method}$

(e) $forName \in E_{VMNative}$
$\Rightarrow classTable \in F_{VMClassLoader}$

(f) $indexOf \in E_{VMNative}$
$\Rightarrow bytes \in F_{java.lang.String}$

(g) $javaVM \in E_{VMJavaVM}$

(h) $e \in E_{VMJavaVM}$
$\Rightarrow mainThread \in F_{VMJavaVM}$

(i) $m \in E_{method}$

Figure 2: Example per-VM constraints that relate native methods to their implementation requirements. Natives can (a) allocate new Java objects (b) use features of VM structures (c) invoke Java virtual methods, (d) invoke Java static methods (f) use fields of Java objects. Default constraints assert certain entities (g) and features (h) to be live. The constraint (i) ensures that if a Java method implementation is live, then its representation in the VM is live.

Figure 2 gives examples of some constraints that handle native method implementations in the class library. These constraints model the fact that native methods can trigger Java-level features such as creating new Java objects and arrays, as well as directly manipulating the VM's internal data structures.

Consider the example constraint (e) in Figure 2, which models the need for the class table, a hashtable that maps strings to class representations in implementing the `Class.forName` Java native method. If this native method is never called (i.e. it never is added to the set $E_{VMNative}$), then the `classTable` pointer need not be analyzed, and consequently, this data structure can be removed.

4.6 Granularity and Natives vs. Sanity

In our experience, writing the constraints for all of Java's bytecodes was comparatively little effort, as this problem is generally well understood and has already been explored in many previous analysis techniques. If we make the assumption that the constant pool entries are resolved and that classes are loaded and initialized, then each bytecode amounts to little more than manipulating Java objects and the stack and performing calls to some simple VM services such as the allocator. At the bytecode level, it is easy to have confidence that our analysis constraints for each bytecode will force the inclusion of the necessary data structures into the image, and that "pure Java" programs will execute without problems on the ExoVM.

However, the bulk of Java—its class library—is not so simple. Java has dozens of classes in its standard library that are wormholes into the VM; many have native methods that manipulate internal VM data structures directly. In our development branch of J9, the VM and the native code that implements the class library are developed separately but significantly interdependent. In the j9cldc class library, there are 75 such native methods, many of which are implemented in assembly code. In the J2SE (j9max) class library, there are more than 200. Some use JNI or internal services to call back into Java code or allocate Java objects. Each of these methods requires constraints that trigger the inclusion of Java code and VM structures that are required to implement them. We were able to derive constraints for many of the most important ones. For some we simply coarsen the granularity of the analysis of data structures and conservatively include some possibly unreachable data structures. Otherwise, we forbid native methods that we do not yet support by dynamically trapping calls to them.

An example of tuning the analysis between fine-grained and coarse-grained is the idea of modeling every pointer in every data structure in the virtual machine as a feature that is only used when certain constraints are triggered, such as the use of a particular native method or VM service. While the most fine-grained approach is attractive because it allows the maximum possible reduction of data structures, only including them under the most specific circumstances, the VM is complex enough that determining the most specific constraints for each pointer becomes infeasible. For many pointers, we were forced to simply assert them either dead or live, depending on whether we intend to support the associated feature in the ExoVM. Asserting them live is always conservative and correct, provided that the data structure that they refer to is correctly identified and copied into the image, but this may bloat the image with data structures unneeded for the particular program. However, asserting these pointers dead may be too aggressive because if the associated language feature or service is needed at runtime, the virtual machine or native libraries will crash due to the missing data structures.

Our approach has taken the middle of the road, asserting many pointers to be dead that correspond to VM features that we do not intend to support, such as dynamic class loading, and asserting some pointers live and always copying the referred data structures because the right constraints may be elusive. Some data structures are always necessary, such as the `VMJavaVM` data structure and the `VMThread` structure for the main java thread. We've developed a suite of micro-programs that target individual features in order to expedite testing and debugging, allowing us to pinpoint the usage of many pointers of VM structures and relate them to language features. Individual tests cover the basic bytecode set of the JVM and target specific native methods and language services. For more complex correctness validation including native methods, we rely on running larger benchmark programs and verifying that each program computes the same results as it does on the complete JVM. An industrial scale, feature-complete implementation of our technique would have to test against the Java language compliance kit, since we do not believe that it is possible to directly prove the correctness of the analysis technique due to the sheer size and complexity of the VM's implementation of native methods and language services.

5. PERSISTENCE

Persistence is the process of taking a snapshot of the fully initialized virtual machine, including the data structures that represent the program and the program's state, and saving it to an image file or other persistent store to be loaded later. Persistence has been studied widely in programming languages and database systems [19] and has a number of compelling advantages for programming systems. Key issues are the transparency and efficiency of the persistence mechanism, as well as data evolution and versioning.

In our system, we perform imaging of the VM only once as part of an offline analysis, so the efficiency considerations do not apply, and we do not support data evolution simply because the kinds of data we are saving are heavily tied to one particular VM implementation. As such, our persistence framework, which we refer to as the *imager*, need not be as general as that in previous systems. After the closure process has computed a set of reachable Java methods, classes, objects, and VM data structures, the imager copies and relocates the data structures that exist inside the virtual machine to a special region of memory which is then saved to the disk. This image file is a compacted snapshot of the VM data structures that represents only the reachable parts of the program. The image file contains essentially a complete ready-to-go VM that can be used immediately by simply mapping it into memory.

5.1 Persisting C-based Data Structures

Once the closure process has computed the set of reachable data structures of the VM that are needed to correctly execute the program, the imager must copy and relocate these data structures to persistent store. These data structures are declared in C but are manipulated by C, C++, and assembly code. The imager therefore needs to persist C data structures in a way that preserves the invariants that are implicit in the code that manipulates them. We began studying the layout of these data structures and the code that manipulates them, discovering that many had implicit and complex structure and invariants. This manual process represents a particularly unromantic but significant amount of our development time, approximately 3-5 man-months. From our efforts we were able to develop a description of each important data structure: its layout, address alignment constraints, contents,

and its pointers to other data structures. The imager uses the description to determine how to copy and relocate VM data structures of each type, which includes computing the size and layout of a particular instance and where pointers to other data structures lie within the structure. This is similar to the description of a Java object that a garbage collector needs in order to scan a Java object for references to other objects, but can be considerably more complicated. We discovered a number of implicit constraints on data structures. Two constraints of note are implicit adjacency/layout requirements, and strangely encoded pointers.

Many kinds of data structures are segregated into segments, which allows mass allocation and deallocation as well as fast traversal over all data structures of a given type. The dependence on this layout is buried deep in the assembly and C code of the VM; to reuse this code without modification requires preserving the invariants it expects. This requires the imager to collect certain structures into new segments during the copy process. Furthermore, some data structures have grown very complex as they evolved over time. For example, the representation of a class has numerous adjacent, embedded members of variable size; code throughout the VM relies on being able to find known structures at computed offsets from the beginning of the structure. Other data structures throughout the VM point into the middle of the class structure. A correct description of this data structure for the imager required a lot of manual analysis of the code to determine its undocumented layout and implicit constraints.

Some pointers are not only pointers, but contain some extra high or low-order bits that are used in implementation tricks for monitors [8], virtual tables, and object headers, etc. These pointers are assumed to point to structures aligned on addresses that are particular powers of two (most often 8, 16, and 256 bytes), which allows the lower bits to be reused. To address this common undocumented tendency, the description of each data structure in the imager contains alignment constraints that are used when the imager chooses a new address for a data structure, making the undocumented constraint explicit. Similarly, pointers that contain extra information bits have special types that instruct the imager to preserve the appropriate low-order bits; the type makes it obvious that the pointer contains extra information. Another problematic feature of the system is the use of *self-relative pointers* within some data structures; a self-relative pointer stores an offset instead of an actual address; instead, code that uses the pointer computes the actual address by adding the pointer's value to the pointer's location. This allows some data structures to be copied to and from disk and shared across processes without relocation. Because the imager moves pieces of these data structures around independently, to reuse the VM code unmodified the imager must encode and decode self-relative pointers while moving data structures. Like pointers with extra bits, self-relative pointers have a special type in the data structure description that documents this fact and allows the imager to handle these pointers with equal ease as normal pointers.

5.2 Compilation

By completely initializing the VM before imaging, the system can also save any compiled code of the application that has been produced by the JIT. In fact, because of the offline nature of the imaging process, we can simply compile all of the reachable methods with the JIT compiler ahead of time. The JIT and its data structures can then be removed completely from the ExoVM, effectively turning the original VM into a static compiler – albeit one which generates superior code because all classes are resolved and initialization code has already been executed. Because the compilation takes place in a closed-world scenario, there is no need to invalidate code and recompile.

The imaging process can support pre-compilation of all the methods in the reachable program by running the JIT compiler after feature analysis and directing it to generate code into the image. Some small modifications to the JIT compiler are necessary to support this; for example, the JIT often writes the absolute address of data structures and functions that it assumes do not move into the compiled code; the imager must make sure that these pointers are found and relocated before the image is finished. We can support this simply by instrumenting the JIT to record where it writes absolute addresses into the compiled code, and then patching the addresses at image load time. With this approach, there is no need to alter the machine code that the JIT generates. This feature was not fully operational due to time constraints, and our experimental results use the interpreter and do not include any compiled code. The size of the compiled code depends on the quality of the JIT compiler and the total amount of reachable code of the program.

5.3 Loading a VM Image

Although the imager is capable of producing an image that contains a complete collection of data structures that represent the program and the VM needed to run the program, the imager is not capable of actually copying the machine code of the VM into the image. The implementation technology of the VM, particularly the linking model of C and C++, precludes this, and computed jumps and branches within machine code cannot be supported without linking information. Our approach to this problem is to separate the data structures (which are stored in the image file) from the *boot VM*, a specialized offline build of the fully featured VM that contains little or no internal data structures. The boot VM lacks the normal VM initialization routines that build these internal data structures, as well as mechanisms such as the JIT compiler and dynamic class loader, but instead only contains VM subsystems that will be needed at runtime for each application, such as the interpreter, garbage collector, natives of the class library, etc. The boot VM loads all of the needed data structures from the image.

Our imager produces image files that are intentionally not relocatable; i.e. all of the internal data structures and code within an image file contain absolute pointers to each other that assume the image starts at a fixed memory address. This simplifies both the imager and the boot VM, allowing the boot VM to simply memory map the image from the file to the specific address and thus begin using the image in memory without relocating any internal pointers. Additionally, the image header contains pointers to the main class, the main method of the program, and to important global VM data structures so that the boot VM need not search the image for where to begin execution.

5.4 Patching and Rebuilding

The separation between the code and data of a VM instance is not perfectly clean, and many internal data structures that are saved in the image contain pointers to internal VM functions that do not exist in the image. The boot VM must supply the implementation of these functions by patching these pointers when the image is loaded into memory.

For example, the VM-level method data structure contains a pointer to code that implements the calling convention for that method when it is called. An interpreted method contains a

pointer to machine code in the interpreter to set up the interpreter state, while a synchronized method has a pointer to code that obtains the lock on its receiver object before executing the method, and so on. When the imager copies a data structure and encounters pointers to VM machine code or a C function, it uses a table of known VM routines to identify the target routine. At load time the boot VM loads the image and replaces these pointers with pointers to its implementation of the corresponding routines.

One further complication with the imaging process is that not all internal data structures can be persisted. In particular, the VM has data structures that correspond to operating-system level resources such as threads that are not transferable from one process to the next. The boot VM rebuilds certain data structures as necessary when it loads the image into memory.

6. EXPERIMENTS

6.1 Footprint

We have implemented pre-initialization, closure, and persistence in a J9-based virtual machine with the j9cldc and j9max class libraries to investigate the memory footprint of the VM and the application in an embedded scenario. These numbers are obtained on the x86 build of J9 running on Linux 2.6. We did not specifically measure the execution time for the imaging process, but even with our completely untuned implementation written mostly in Java and running in interpreted mode, the entire load, initialize, closure, and copy process of the ExoVM took less than 5 seconds on a fast Pentium IV workstation for all our benchmarks.

To evaluate the effectiveness of the ExoVM approach, we measured a number of footprint factors for our benchmark programs. First, we evaluate the fixed cost of the VM in terms of the VM's static code and data footprint for the two original VM configurations and the ExoVM specialized boot VM. The j9cldc configuration consists of 600kb of compiled VM code and natives, 260k of read-only data (of which 190kb is the class library compiled into the executable), 20kb of initialized data and 17kb of uninitialized data. The j9max configuration consists of 750kb of compiled VM code and natives, 90kb of read only data, 25kb or initialized data, and 17kb of uninitialized data. To reduce the size of the boot VM, we statically compiled out some subsystems, including the JIT compiler, bytecode parser and verifier, zip library support, and some initialization routines, saving about 200kb of compiled code. We believe that there is more code that can be removed from this specialized VM, but linking issues and time constraints limited our ability to explore this.

In Figure 3, we compare the dynamic memory footprint measurements for the data structures and loaded classes across our benchmarks for the j9cldc configuration. The first row of each benchmark contains the measurements of several footprint factors on the unmodified VM running the applications with the j9cldc class library. These footprint factors are CLIB; the size of the j9cldc class library which is compiled into the binary executable; ROCL, or read-only portions of the application classes (`VMROMClasses`); RWCL, or the read-write portions of these same application classes (`VMClasses`); NHA, or non-heap allocations, which are data structures allocated by the VM that are not Java objects and thus not part of the heap. Each of these numbers is given in kilobytes. The two remaining footprint factors apply only to ExoVM images. These are INH, or imaged non-heap data structures, which are non-heap data structures that were allocated during pre-initialization and have been persisted; and IHEAP, which is the initial heap of Java objects, consisting of everything from string constants to application objects that have been determined to be reachable by the closure process. Note that we do not measure the heap of the program here; we were able to successfully execute the benchmarks with just 128kb of heap (except kXML, which required 512kb), which makes the VM data structures by far the dominating factor.

These measurements show the effectiveness of pre-initializing the virtual machine and the application. With a completely built image, the ExoVM has no need of an external class library (CLIB). Feature analysis detects that a number of classes are unused and removes them, showing a moderate reduction in the size of the read-write class representations (RWCL). The size of the initial heap (IHEAP) generated by running the class initializers in the virtual machine is relatively small. But by far the

J9CLDC Dynamic Footprint

	CLIB	ROCL	RWCL	INH	NHA	IHEAP	total
Chess	188	74	60	0	394	0	716
-exo	0	113	42	33	10	14	212
Crypto	188	70	62	0	466	0	786
-exo	0	114	46	25	10	41	236
kXML	188	56	58	0	483	0	785
-exo	0	113	45	27	11	50	246
Parallel	188	49	45	0	415	0	697
-exo	0	87	26	30	89	33	265
PNG	188	26	48	0	383	0	645
-exo	0	74	32	23	11	30	170
RegExp	188	47	55	0	389	0	679
-exo	0	98	41	26	11	21	197

Figure 3 shows dynamic non-heap memory footprint for six benchmarks on the j9cldc configuration. Each benchmark has two rows: one for its footprint in the standard VM, and the next row for its footprint using the ExoVM system.

J9MAX Dynamic Footprint

	CLIB	ROCL	RWCL	INH	NHA	IHEAP	total
Chess	0	597	162	0	557	0	1316
-exo	0	619	172	27	10	171	999
Crypto	0	595	163	0	591	0	1349
-exo	0	615	173	26	10	195	1019
kXML	0	588	159	0	646	0	1393
-exo	0	610	169	26	11	204	1020
Parallel	0	574	147	0	549	0	1270
-exo	0	0	0	0	0	0	0
PNG	0	549	148	0	504	0	1201
-exo	0	577	160	25	11	181	954
RegExp	0	571	156	0	518	0	1245
-exo	0	598	168	26	11	173	976

Figure 4 shows dynamic non-heap memory footprint for six benchmarks on the j9max configuration. Each benchmark has two rows: one for its footprint in the standard VM, and the next row for its footprint using the ExoVM system.

biggest factor is the reduction of the VM's dynamic non-heap memory allocations. This shows that pre-initialization of the VM and feature analysis allow the ExoVM to remove the dominant factor of space consumption in these benchmarks. The reduction of nonheap memory allocations is between 62 and 73% for these six benchmark applications.

Figure 4 evaluates the ExoVM system over the j9max configuration, which consists of the same VM, but a more complex, fully featured class library. In this scenario, the class library is much larger and not compiled directly into the virtual machine's binary. However, we can see that the dominant cost is now the size of loaded classes, because the more fully featured class library has many more interdependencies that force many classes to be loaded and initialized.

The most surprising result is that running the feature analysis to produce an image for each of these programs does not yield a smaller ROM or RAM class footprint. We investigated the reason for this and discovered that the j9max's `Class.getName()` implementation uses a `HashMap` that maps a class representation to its `String` name. Because our analysis is partly written in Java and runs on this underlying class library to compute the closure, if the program being analyzed calls the `Class.getName()` method, then the analyzer will discover that this `HashMap` is reachable, and begin analyzing its contents. Because these classes are reachable through Java references, it therefore concludes that all loaded classes are live, and none are removed from the image.

We were not able to successfully run the Parallel benchmark on the ExoVM because the larger class library demanded an implementation of protection domains. This highlights another problem with a larger class library. Adding a security layer tends to demand reflective features from the VM that thwart our program analysis.

6.2 Feature study

During the course of developing the ExoVM system and testing feature analysis for correctness, we wrote a large number of Java micro-programs that each uses a specific language feature, such as virtual dispatch, throwing an exception, calling API methods, running threads, etc. While primarily intended for our internal use in testing correctness, they had the side effect of exposing just how much of the class library and VM is tied to a particular language feature. Though we don't claim that our micro-program suite is fully comprehensive of the Java language, it did highlight important issues.

Image Size: microprograms

	cldc	max
empty	5	5
arrays	38	225
checkcast	42	228
constructors	13	31
floating point	7	7
nullptr	13	31
.getClass()	30	872
refarray	40	226
Hello world	75	872

Figure 5

Our micro-programs were all less than 25 lines of code and primarily target a single language-level feature. We found a good approximation of the cost of a feature to be the size of the image generated by our analysis, which includes not only VM data structures, but also persisted classes and objects. As a starting point, we tested how small an image our system could generate for the empty program; i.e. a single static main method that just returns. On both j9cldc and j9max, our system generates a 5kb image that contains the main class (1kb), `java.lang.Object` (1kb), the `VMJavaVM` structure (1.3kb), a thread (0.6kb), and a small number of other data structures. This is enough to reuse the existing VM code unmodified and execute successfully.

From this starting point, we investigated the incremental cost of supporting individual languages features; Figure 5 shows several microprograms and the resulting image size for the j9cldc and j9max configurations. From the table, we can see that several of the programs that generate small images on the j9cldc configuration have large images on j9max.

We were able to pinpoint the problems that cause this phenomenon of "feature explosion" in j9max by using these unit feature tests. Our analysis revealed that the larger class library contains a small number of "precarious" dependencies, such as the `HashMap` in the `Class.getName()` implementation mentioned previously. When one such dependency is triggered, it tends to pull in a large subset of the class library as a whole. This can be seen in the tests that construct and print exceptions: they tend to pull in a large portion of the class library, which ultimately dwarfs their small size. Our conclusion from this study is that future design of class libraries and careful implementations should strive for modularity in features so as to avoid penalizing small programs and avoid precarious dependencies. Another approach might be to embed more special knowledge into analysis about the Java-level entities that implement Java features, such as introducing a special case for the `Class.getName()`'s internal data structures. This remains as future work.

6.3 Experience

In our experience developing the ExoVM system in J9, we learned important specific lessons about its implementation and virtual machine design in general that we think are valuable to others. The first is that complex and arcane data structures frustrate automated imaging techniques, and judging from the implementation complexity that seems to replicate itself over and over throughout the virtual machine, we simply do not believe they are worth whatever gain they intend. By far most of our manual effort was inferring implicit constraints of data structures and fixing problems with pointers and layout tricks, working backwards from VM crashes. Although certain techniques have advantages for performance or space usage, our overwhelming sense after studying the code is that the most complicated data structures have evolved by accretion and their deep entanglement with the VM makes them particularly dangerous to migrate or refactor. We think that our work shows the value of persisting the internal VM data structures for an embedded domain, and simpler, more regular data structures make this technique far easier.

The second lesson we learned from our experience is that there appears to be more modularity to source-level language features than previously thought. This dimension of modularity does not seem to be borne out in current virtual machine design and class library implementations, including J9 and those with which the authors have previous experience. We believe that this dimension of modularity has important applications in the embedded domain, and that valuing it more highly in the design of new virtual machines will have positive consequences for the ability to scale from small devices to server class machines.

The third lesson that we learned is that the implementation technology of the virtual machine itself matters considerably. We

cannot achieve our ultimate goal of total automatic VM specialization given J9's current implementation technology, in particular the static linking model inherent in C and C++ applications. A large amount of our development effort has been spent in recovering implicit usage patterns of data structures in the virtual machine which is difficult to automate in these languages. Given our experience with large applications written in higher-level, statically typed languages like Java, we believe that much of this analysis can be streamlined, if not automated completely, if the VM itself were implemented in a language that is more amenable to disciplined program analysis.

Not surprisingly, we found that complexity of the class library makes an important difference to the footprint of an application, especially with the implementation of the basic language features such as exceptions. The CLDC implementation of the class library contains not only fewer classes over all, but the implementation of basic classes such as exceptions has fewer dependencies, resulting in smaller image sizes. The difference between the CLDC exceptions and those in j9max is many more live classes and consequently more used language features. Further, the implementation of exceptions and I/O (particularly international formatting of strings) is significantly more complex in the j9max library. For this technique to work well on such class libraries, more modularity in these implementations seems to be necessary, or the analysis must be improved.

Java's dynamic invocation of class initializers may work well for a bigger domain, but our results with pre-initialization of the classes in an image tends to suggest that for this domain, significant gains can be made by changing the model.

7. CONCLUSION

We believe the investigation of feature analysis contributes positively to a grand challenge in virtual machine construction: the design and implementation of a language runtime and compilation model that seamlessly adapts across static and dynamic views of compilation and scales from extremely small systems up to very large systems. Our experimental results show that pre-initialization coupled with feature analysis can reduce the non-heap footprint of the java virtual machine's data structures by as much as 73% and the VM code size by as much as 30% by removing unnecessary subsystems.

This work also has wider applicability because it can provide the basis for relating language features to their efficiency considerations more directly. We illustrated how feature analysis has shed light on the interconnectedness of the virtual machine and the class library implementation with constraints. We believe that this is just a first step to exposing the efficiency implications of feature use to application developers to whom footprint matters, such as embedded system programmers.

8. FUTURE WORK

Our VM persisting techniques are an artifact of the implementation technology of the virtual machine we chose for our research. We believe that much more would be possible if we could automate the derivation of constraints and the persistence mechanism.

We believe there are more opportunities for static optimization such as in an ideally static closed-world scenario, where the imaging process might be able to copy both the application's code, the internal data structures of the VM, and also the live code of the VM into the image, producing a completely customized VM compiled together with the application into a standalone program. This would allow the VM and its JIT compiler to be reused as a static compilation system, perhaps allowing it to employ sophisticated compiler optimizations like partial evaluation or static specialization to itself and the application code together.

Conversely, we believe this work might have applicability for dynamic languages as well, where in a dynamic open-world scenario, a more flexible VM infrastructure that was decomposed modularly according the features of the language might employ a dynamic feature analysis so that parts of the program and VM infrastructure are loaded on demand as they are needed by the program. The VM might reduce the granularity of dynamic loading to single methods rather than single classes, only loading methods as they are used. Similarly, the VM might defer the construction of internal data structures until they are demanded by the first use of a particular programming language feature. This may significantly improve performance for small dynamic programs and help combat large class libraries.

Shorter-term work in this area would be to further extend the development of the analysis technique to larger sets of features and to a more powerful runtime and JIT compiler.

9. REFERENCES

[1] C. Ananian and M. Rinard. Data Size Optimizations for Java Programs. In *2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*. San Diego, CA. June 2003.

[2] D. Bacon and P. Sweeney. Fast Static Analysis of C++ Virtual Calls. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96).* San Jose, CA. Oct. 1996.

[3] G. Chen, M. Kandemir, N. Vijaykrishnan, M. Irwin, B. Mathiske, and M. Wolczko. Heap Compression for Memory-constrained Java Environments. In *Proceedings of the 18th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '03).* Anaheim, CA. Oct 2003.

[4] Connected Limited Device Configuration (CLDC). `http://java.sun.com/j2me`

[5] A. Courbot, G. Grimaud, and J.-J. Vandewalle. Romization: Early Deployment and Customization of Java Systems for Constrained Devices. In *Proceedings of Second International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS).* Nice, France, Mar 2005.

[6] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *the 9th European Conference on Object-Oriented Programming (ECOOP '95).* Aarhus, Denmark. Aug. 1995.

[7] J. Koshy and R. Pandey. VM*: A Scalable Runtime Environment for Sensor Networks. In *The 3rd annual conference on Embedded Network Sensor Systems (SENSYS '05)*. San Diego, CA. Nov. 2005.

[8] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems,*

Languages and Applications (OOPSLA '99). New York, New York. Nov. 1999.

[9] D. Spoonhower, J. Auerbach, D. Bacon, P. Cheng, and D. Grove. Eventrons: A Safe Programming Construct for High-Frequency Hard Real-Time Applications. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI '06)* Ottawa, CN. June 2006.

[10] F. Tip, C. Laffra, P. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*. New York, New York. Nov. 1999.

[11] F. Tip and J. Palsberg. Scalable Propagation-based Call Graph Construction Algorithms. In *the 15th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. Minneapolis, MN. Oct. 2000.

[12] B. L. Titzer. Virgil: Objects on the Head of a Pin. In *Proceedings of the 21th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '06)*. Portland, Oregon. Oct. 2006.

[13] Sun Microsystems, J2ME Building Blocks for Mobile Devices, 2000.

[14] D. Rayside and K. Kontogiannis, Extracting Java library subsets for deployment on embedded systems," Sci. Comput. Program., vol. 45, no. 2-3, pp. 245-270, 2002.

[15] Z. Chen, Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison-Wesley Longman Publishing Co., Inc., 2000.

[16] D. Rayside, E. Mamas, and E. Hons, Compact java binaries for embedded systems," in Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, p. 9, IBM Press, 1999.

[17] D.-W. Chang and R.-C. Chang, Ejvm: an economic java run-time environment for embedded devices," Software Practice & Experience, vol. 31, no. 2, pp. 129-146, 2001.

[18] D. Mulchandani, Java for embedded systems," Internet Computing, IEEE, vol. 2, no. 3, pp. 30-39, 1998.

[19] M. P. Atkinson, M. Dmitriev, C. Hamilton, T. Printezis: Scalable and Recoverable Implementation of Object Evolution for the PJama1 Platform. POS 2000: 292-314.